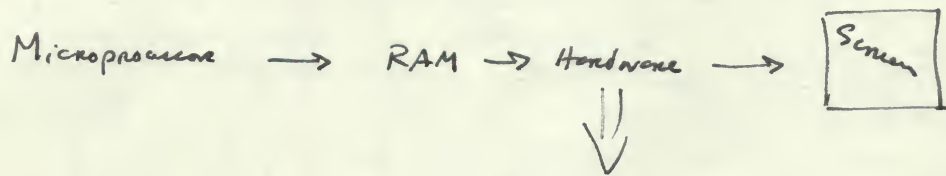


Display Lists

PIXEL - PICTURE Element



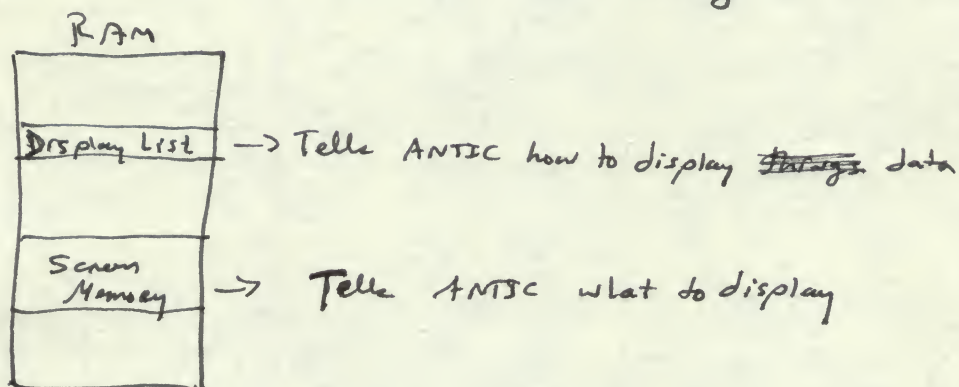
Atari: This is done by 2 chips (special purpose)

Display Processor — ANTIC

Grab a byte + put it out — CTIA

ANTIC's Program = Display List

Data = Screen Memory

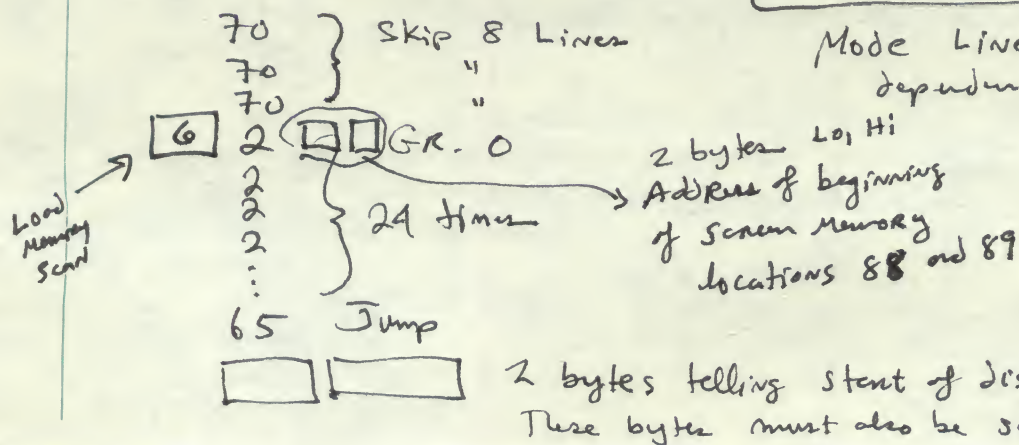


Display List — Sequence of instructions telling ANTIC how to display something on the screen. Interprets screen memory data.

DISPLAY LIST

TV = 192 Horizontal Scan Lines

Mode Line = Collection of scan lines dependent on graphics mode



SAMPLE PROGRAMS

- To look at a display list:
GR.0: A=PEEK(560)+256*PEEK(561)
FOR I=0 TO 31: PRINT PEEK(A+I); " ";: NEXT I
(for other graphics modes, replace 31 with other values)
- To mindlessly modify this display list:
GR.0: A=PEEK(560)+256*PEEK(561)
FOR I=3 TO 14: POKE A+2*I, I: NEXT I
A is the address of the beginning of the display list
- To insert one odd line and write to it:
GR.7: POKE 559, 0: A=PEEK(560)+256*PEEK(561)
POKE A+8, 7: POKE A+9, 7: FOR I=85 TO 93: POKE A+I-14, PEEK(A+I): NEXT I
POKE 559, 34: B=PEEK(A+4)+256*PEEK(A+5)
POKE B+122, 33: POKE B+123, 34: POKE B+125, 99: POKE B+146, 163: POKE B+147, 225
COLOR 2: PLOT 30, 4: DRAWTO 90, 4: DRAWTO 90, 63: DRAWTO 30, 63: DRAWTO 30, 5

Display List Instruction Table (Abridged. All values in decimal)

How much memory is allocated for screen memory

Antic	Value	What it does	Lines per Pixel	Pixels per row	Colors	Bytes per Line	BASIC GR. mode	BASIC "A" value	BASIC "B" value	C	D	E
	0	blank 1 line	—	—	—	—	—	—	—	—	—	—
	1	jump (3 byte instruction)	—	—	—	—	—	—	—	—	—	—
	2	Character mode	8	40	2	40	0	32	0	960	—	—
	3	"	10	40	2	40	none	—	—	—	—	—
	4	"	8	40	4	40	none	—	—	—	—	—
	5	"	16	40	4	40	none	—	—	—	—	—
	6	"	8	20	5	20	1	34	0	400	80	160
	7	"	16	20	5	20	2	24	0	200	40	160
	8	Map mode (colored squares)	8	40	4	10	3	34	0	200	40	160
	9	"	4	80	2	10	4	54	0	400	80	160
A	10	"	4	80	4	20	5	54	0	800	160	160
B	11	"	2	160	2	20	6	94	0	1600	320	160
C	12	"	1	160	2	20	none	—	—	—	—	—
D	13	"	2	160	4	40	7	94	96	3200	640	160
E	14	"	1	160	4	40	none	—	—	—	—	—
F	15	"	1	320	2	40	8	176	80	6400	1280	160
10	16	Blank 2 lines	—	—	—	—	—	—	—	—	—	—
20	32	Blank 3 lines	—	—	—	—	—	—	—	—	—	—
30	48	Blank 4 lines	—	—	—	—	—	—	—	—	—	—
40	64	Blank 5 lines	—	—	—	—	—	—	—	—	—	—
50	80	Blank 6 lines	—	—	—	—	—	—	—	—	—	—
60	96	Blank 7 lines	—	—	—	—	—	—	—	—	—	—
70	112	Blank 8 lines	—	—	—	—	—	—	—	—	—	—
41	65	Imp + wait for v.blank (3 byte instruction)	—	—	—	—	—	—	—	—	—	—

Definitions: (these only apply to displays written by the BASIC cartridge.)

A: length of display list B: number of unused bytes between end of display list and beginning of memory map
C: length of memory map D: number of unused bytes between end of display list and beginning of text window map
E: length of text window map

Indirection

Note an efficiency - the more times a statement is executed, the more efficient

6502 - 2 levels of indirections

Indirection makes the kind of numbers you get controllable by calculations -

You can calculate on the middle "man" which will then point to the ~~actual value~~ contents of the actual value

One Example - Color on the Atari - Indirect Color System

Color Register indirectly controls colors set up by the setcolor command

DISPLAY LIST INTERRUPT (DLI) can catch the beam between any two scan lines and change colors

Allows calculations and numbers "gone" with color & brightness

Character Sets

CHBASE = LOC. 756

Contents => Page no. of

Dot matrix - 8x8

start of char. set
page = 256 bytes

(if you use DLI's you need the ANTIC CHBASE location)

A character set uses up 512 bytes of RAM

BASIC

See IRIDIS FONT Program (FONTEDIT)

WHAT IS INDIRECT?

You refer not to the thing itself, but to something that refers to the thing

Levels of indirection (and flexibility):

Low	Middle	High
"Get a 4"	"Get the number in address 21379"	"Get the number in the address which is given in addresses 81 and 82"

✱

A SIMPLE EXAMPLE OF INDIRECT: COLOR REGISTERS

DIRECT:
"Put RED on the screen"


INDIRECT:
"Put the color in register 2 on the screen"

Why is indirect better? Because "one number in register 2 controls color for many pixels.
Specifically:

- 1) less memory consumption (2 bits for 4 regs instead of 4 bits for 16 colors)
- 2) more colors ~~and~~ to choose from (128 vs 16)
- 3) fast time control of screen colors
- 4) DLI capability
- 5) calculations + numbers games with colors. (luminosity, for example)


CHARACTER SETS

DIRECT

65 \Rightarrow A \Rightarrow 

↑
this collection of dots comes from ROM

INDIRECT

65 \Rightarrow (CHBASE) \Rightarrow A \Rightarrow 

↑
this collection of dots comes from ROM or RAM, depending on where CHBASE points

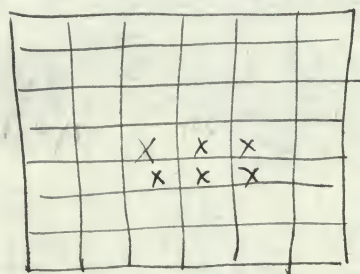
Why is indirect better? Because one number in CHBASE defines the character set for many characters.

Advantages:

- 1) Multiple character sets (Fonts). Available on very short notice (US)
- 2) Graphics character sets
- 3) DLI capability
- 4) numbers games ~~and~~ with character sets.

CHBASE is at 756 (shadow)

Conventional Playfield Systems



Screen
2-dimensional

RAM
one-dimensional

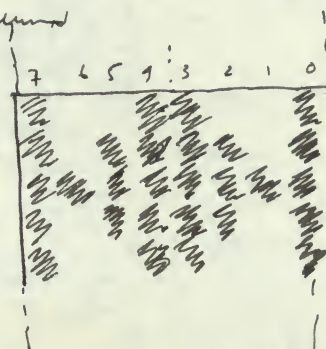
To get 2 dimensional effects out of one-dimensional data arrays, you must calculate (particularly multiply) all the new positions which is slow. And, ~~while~~ ^{while} the image is contiguous on the screen, it is not contiguous in RAM.

Atari Solution: Player-Missile Graphics

Player = Table of bytes which controls a whole column on the screen, placed on top of whatever else is there.

The player table is 8 bits (one byte) wide and affects ~~the~~ an entire column

For Example:



PLAYER TABLE

$10011001 = 99$
 $10111101 = BD$
 $11111111 = FF$
 $10111101 = BD$
 $10011001 = 99$

128 or 256 bytes
long

Horizontal Motion controlled by horizontal position

Vertical Motion controlled by moving images up & down in the table
independent

independent
Player tables, each have its own color register

Each player has a "2-bit" missile attached to it

Image priorities - controllable

Lecture Outline

Player-Missile Graphics

Problem: Animation (moving images)

Traditional solution: playfield animation

Weaknesses: 2d image, 1d RAM. Bit sliding.

Therefore slow or small or simple motion.

Solution: Player-missile graphics

Fundamental idea: 1-d in RAM, 1d (sort of) on screen.

Map table of bytes directly onto screen, on top of playfield.

Draw an image bit by bit.

Animation with player-missile graphics

Vertical motion: one-dimensional move routine (not hard)

Horizontal motion: direct control of horizontal position register. (ridiculously easy)

Embellishments:

- 4 players, each with its own color register

- Controllable player width (still 8 bits of resolution)

- Two resolutions

- Missiles

- Image priorities

How to do it: Sample program

Potential:

- Animated players

- Additional color

- Special off-line characters

- Cursors

512 low byte of DLI vector
 513 high byte of DLI vector
 559 a 62 here gives 1-line resolution, a 46 gives 2-line
 704 color of player-missile 0
 705 color of player-missile 1
 706 color of player-missile 2
 707 color of player-missile 3
 53248 horizontal position of player 0
 53249 horizontal position of player 1
 53250 horizontal position of player 2
 53251 horizontal position of player 3
 53252 horizontal position of missile 0
 53253 horizontal position of missile 1
 53254 horizontal position of missile 2
 53255 horizontal position of missile 3
 53256 size of player 0 (0=normal, 1=double, 3=quadruple)
 53257 size of player 1 (0=normal, 1=double, 3=quadruple)
 53258 size of player 2 (0=normal, 1=double, 3=quadruple)
 53259 size of player 3 (0=normal, 1=double, 3=quadruple)
 53266 color of player-missile 0 (for DLI's only!)
 53267 color of player-missile 1 (for DLI's only!)
 53268 color of player-missile 2 (for DLI's only!)
 53269 color of player-missile 3 (for DLI's only!)
 53270 color of playfield 0 (for DLI's only!)
 53271 color of playfield 1 (for DLI's only!)
 53272 color of playfield 2 (for DLI's only!)
 53273 color of playfield 3 (for DLI's only!)
 53274 color of background (for DLI's only!)
 53275 sets player/playfield/background priorities (only 1 bit on!)
 53277 put a 3 here to enable player-missile graphics
 54279 put high byte of PMBASE here
 54286 put a 192 here to enable DLI's

List doesn't include all shadow registers

```

10 SETCOLOR 2,0,0:X=120:Y=48:REM Set background color and player position
20 A=PEEK(106)-8:POKE 54279,A:PMBASE=256*A:REM Set player-missile address
30 POKE 559,46:POKE 53277,3:REM Enable PM graphics with 2-line resolution
40 POKE 53248,X:REM Set horizontal position
50 FOR I=PMBASE+512 TO PMBASE+640:POKE I,0:NEXT I:REM Clear out player first
60 POKE 704,216:REM Set color to green
70 FOR I=PMBASE+512+Y TO PMBASE+516+Y:READ A:POKE I,A:NEXT I:REM Draw player
80 DATA 153,189,255,189,153
90 REM Now comes the motion routine
100 A=STICK(0):IF A=15 THEN GOTO 100
110 IF A=11 THEN X=X-1:POKE 53248,X
120 IF A=7 THEN X=X+1:POKE 53248,X
130 IF A=13 THEN FOR I=6 TO 0 STEP -1:POKE PMBASE+512+Y+I,PEEK(PMBASE+511+Y+I):NEXT I:Y=Y+1
140 IF A=14 THEN FOR I=0 TO 6:POKE PMBASE+511+Y+I,PEEK(PMBASE+512+Y+I):NEXT I:Y=Y-1
150 GOTO 100
  
```

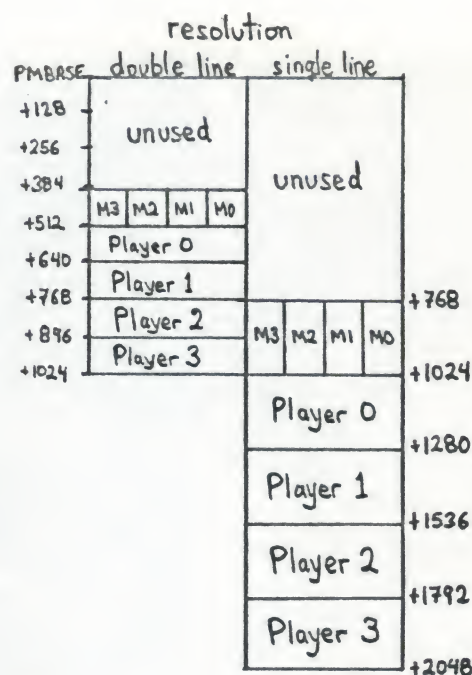
Assume G.R.O

```

2 REM Display List Interrupt Demo Program
3 REM (might not work every time---reset and try again if it fails)
4 REM
  
```

```

10 A=PEEK(560)+256*PEEK(561)+15:REM Find address of middle display list byte
20 POKE A,130:REM Put in a display list interrupt instruction
30 FOR I=1 TO 16:READ A:POKE 1536+I,A:NEXT I:REM Set up DLI service routine
40 DATA 72,169,80,141,10,212,141,23,208,169,88,141,24,208,104,64 (Change Foreground + Background color registers)
50 POKE 512,1:POKE 513,6:REM Poke in interrupt vector (These values in 512 + 513 are the 60, 141)
60 POKE 54286,192:REM Enable DLI's
  
```



Player-Missile Graphics RAM Positioning
 (PMBASE must be on 1K boundary for
~~single~~ double-line resolution, 2K boundary
 for single-line resolution.)

*PEEK of 106 = how many pages
of RAM available*

Display List Interrupts — Value when integrated with Player Missile Graphics, Color Registers, etc.

First Point: Screen is not a static display - it is a timed sequence

DLI — Allows you to cut into the screen refresh at a controllable position (vertically).

Display List controls format of the screen

Display List mode has a 2 byte code

e.g. $\left. \begin{array}{c} 02 \\ 07 \\ 07 \\ 07 \\ 02 \\ 02 \\ 04 \end{array} \right\} \text{HEX}$

If bit 7 of the DLM BYTE is set to 1, the DLM then calls for an interrupt.

e.g. $\left. \begin{array}{c} 82 \\ \equiv \\ 87 \\ 87 \\ 87 \end{array} \right\} \text{HEX}$

on 8 as the left digit sets bit 7

Atroc is told to interrupt the 6502 which causes the 6502 to make a change in Atroc's Display List data before Atroc executes its next instruction.

All this requires that you write Display List Interrupt Service Routines (DLISR) to tell

the 6502 what to change.

The DLI causes 6502 to look in a vector address at HEX 200 and 201 (512, 513 decimal) to find out where the DLISRs are located.

Finally, an enable bit must be set (54286 decimal) for the whole thing to work

What can you do with a DLI?

You can change any register in antic, mid screen

You can change { colors
Player positions } mid screen
Character sets
Player colors
Player widths
Player play field priorities

Shadowing

In Antic, there are many registers.

Many of these registers have copies in RAM.
(This called shadowing; for many registers
the O.S. shadows registers automatically)

So, if you are doing DLI's, you must
put them in the ANTIC registers, not the
O.S. shadow registers.

Maximum of 9 colors on one line.
5 Play fields + 4 Players

NOTE: This system has a vertical
architecture in graphics program design.

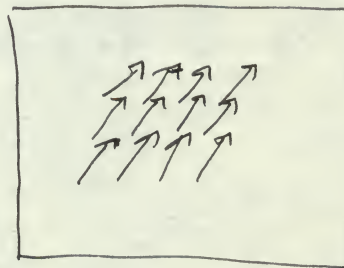
Scrolling -

Cosmic Scrolling

- Moving character from one pixel to another.

Whole picture jumps around

Assembly routine to move all locations in the screen memory to different locations

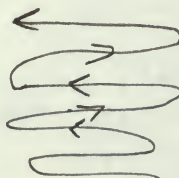


On the Atari, you can move ~~the~~ the playfield by changing the LMS (Local Memory Scan) bytes in the display list

70
70
70
42 LMS
2
2
2
2

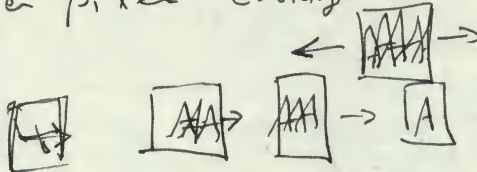
Usually, you only set the LMS byte once in a display list - but you could set it differently for each row line

This causes a "smacking" effect



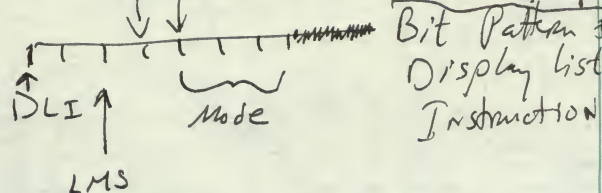
Fine Scrolling

Moving a character around within a pixel (horizontally or vertically)



The values for these scroll bits should not be larger than the size of a pixel for a given mode.

Display List Instruction



Display list bit pattern - Set these bits for all lines you wish to scroll

Playfield width? Problem of what to do about the playfield when scrolling.

3 Playfield widths - narrow, normal & extra wide.

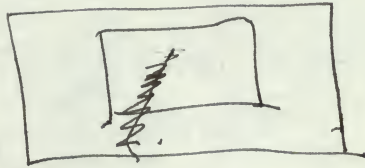
Bits D_0 & D_1 define the width of the playfield.

~~Playfield width must be set to accommodate scrolling.~~

Playfield width must be set to accommodate scrolling.

Best Application for ^{fine} scrolling - sliding things ^(e.g. large images) around.
Combining fine scrolling & coarse scrolling
(i.e. both within & between characters).

Example: Key scrolling to joystick



To scroll vertically, you have to add an entire line's worth of characters when you reach the end of the line.

Note: Fine scrolling controls the pixel, not the character.

~~scrolling~~
Few people have used scrolling

To write a program to Fine/Coarse Scroll

In assembly language -

Fine scroll across ~~the~~ the width of a character
and change the offset the LMS bytes. When
you have reached the end of one character, you
coarse scroll.

Fine scrolling allows you to create a
window on a large "img" and move it
around. In other words, you can
have a window into a large image.

Also Menus would be a good application for
scrolling.

See Chris's "Monday" "Module" on scrolling
available next Monday.

ATARI

BIZARRE

The Numbers + Other Symbols

Char	CTRL	Orange Code C+32	Green Code C	Blue Code C+160	Pink Code C+128
Space	♡	32	0	160	128
!	†	33	1	161	129
"	‡	34	2	162	130
#	§	35	3	163	131
\$	¶	36	4	164	132
%	‖	37	5	165	133
&	∕	38	6	166	134
'	∕	39	7	167	135
(Δ	40	8	168	136
)	◻	41	9	169	137
*	◻	42	10	170	138
+	◻	43	11	171	139
,		44	12	172	140
-		45	13	173	141
.		46	14	174	142
/	◻	47	15	175	143
0	◻	48	16	176	144
1	◻	49	17	177	145
2	◻	50	18	178	146
3	◻	51	19	179	147
4	◻	52	20	180	148
5	◻	53	21	181	149
6	◻	54	22	182	150
7	◻	55	23	183	151
8	◻	56	24	184	152
9	◻	57	25	185	153
:	◻	58	26	186	154
;	◻	59	27	187	155 (CR/LF)
<	↖	60	28	188	156
=	↗	61	29	189	157
>	↘	62	30	190	158
?	↙	63	31	191	159






ATARI

BIZARRE!

The weird, weird world of Atari Character Graphics
in GR.1 and GR.2 modes

The following ~~characters~~ codes correspond to the Alphabet in the
default colors. Noted:

The Alphabet

Character/ letter	Orange Code	Green code + 32	Blue code + 128	Pink code + 160
@	64	96	192	224
A	65	97	193	225
B	66	98	194	226
C	67	99	195	227
D	68	100	196	228
E	69	101	197	229
F	70	102	198	230
G	71	103	199	231
H	72	104	200	232
I	73	105	201	233
J	74	106	202	234
K	75	107	203	235
L	76	108	204	236
M	77	109	205	237
N	78	110	206	238
O	79	111	207	239
P	80	112	208	240
Q	81	113	209	241
R	82	114	210	242
S	83	115	211	243
T	84	116	212	244
U	85	117	213	245
V	86	118	214	246
W	87	119	215	247
X	88	120	216	248
Y	89	121	217	249
Z	90	122	218	250
[91	123	219	251
\	92	124	220	252
]	93	Clears Screen	221	253
^	94	126	222	254
_	95	127	223	255
				

NOTE LOCATION 756 controls upper/lower case

Poke 756, 224 for upper case

Poke 756, 226 for lower case

Note: CLR Screen in
lower case fills the
screen with orange
hexes

Program to mix text with Graphics level 8.

```
80 DIM A$(40)
100 GR.8
200 I0 = PEEK(560) + PEEK(561)*256
220 I1 = PEEK(I0+4) + PEEK(I0+5)*256
240 INPUT A$: IF LEN(A$) = 0 THEN 240
300 PRINT "X POSITION"; INPUT PX
310 PRINT "Y POSITION"; INPUT PY
380 FOR U=1 TO LEN(A$)
390 I2 = 57344 + ((ASC(A$(U,U)) - 32)*8)
400 I3 = I1 + PY*40 + PX + U - 1
410 FOR Z=0 TO 7: POKE I3 + Z*40, PEEK(I2) (I2)
420 POKE I3 + Z*40, PEEK(I2 + Z)
440 NEXT Z
460 NEXT U
500 GO TO 240
```

{ ask for any char string
ask for X, Y location in gr:8 (320x192).
places whatever is in A\$ at location
(X, Y) while in Graphics 8. }